SEPG
FEDERAL AVIATION ADMIN
INTERFACE
Newsletter of the Software Engineering Process Group
Volume 5, Number 4 • November 1996

Software Testing of Safety Critical Systems – Part 1

by Patrick Brown

"The number of software defects found in test is a pretty good indication of the number remaining." Watts Humphrey of the Software Engineering Institute found that his students (inadvertently) injected one defect for every ten source lines of code (SLOC). Only about half of these defects were found in test. In 1986, Herbert Hecht reported better results: for every million SLOC, there were 20,000 bugs and 90% were found by conventional testing, so ***only*** 2000 remained. Even by this more optimistic measure (using experienced programmers), large real-time software systems can expect a minimum of two bugs remaining after test for each KSLOC.

For safety-critical systems, that's bad news. It raises hard questions about the trustworthiness of software, the faith we put in testing, and the methodologies we use for ensuring high quality software. Accidents due to latent software faults, which have become manifest sometimes years after a system has gone into operation, are a ready confirmation of these statistics. This was the case with an X-ray treatment device (the Therac-25) in which 6 people received massive radiation overdoses before the problems were found. A keen awareness of the need for greater caution in developing safety-critical software and the resulting research has led to some insights and improved approaches.

As much as we may try, testing can never be expected to remove all defects in non-trivial software systems. As an example, the Space Shuttle's flight software is extraordinarily carefully developed and tested. But faults have slipped through and been triggered during a mission by an unexpected sequence of inputs.

Software allows a very large number of possible inputs and an even larger number of states. Exhaustive testing is impractical or impossible, at least within the remaining lifetime of the universe. Software also exhibits properties of chaos theory, where minuscule changes in inputs can trigger huge and almost unpredictable changes in output. Therefore, "boundary" value testing or "range" testing of each input separately, while necessary, is not sufficient. Because software is inherently non-linear as compared with older, largely linear analog systems, testing alone cannot ensure the quality or ***safety*** of a software-controlled system.

Some computer scientists would argue that development testing is largely wasteful and that the effort is better spent on other parts of the development process or on formal, mathematically based methods. Typically, development testing is used to show that the software satisfies requirements found in a software requirements specification. With a restricted set of inputs in an artificially constrained, "standard" environment, testing verifies that the software responds as required or expected. Commonly, during test in the absence of a formal development methodology, lots of errors are uncovered and fixed. Since many of those errors turn out to be requirements errors, such testing serves a necessary and valuable purpose which is not entirely eliminated by using formal methods.

Why isn't development testing nearly enough? First, requirements contain mistakes because of misunderstandings and ambiguities in communication among users, designers, programmers, and relevant others. Nancy Leveson, in her book ***Safeware - System Safety and Computers***, ©1995, states that almost all safety-related software failures are due to the implementation of wrong requirements. Second, requirement specifications often cannot capture the richness of the user's concept or the operational environment.

No amount of verification testing against requirements will reveal these problems. This narrowly focused testing usually does not rule out ***unintended*** responses, that is, software responding in unexpected ways to unplanned input. For this, there are other types of testing, including operational T&E and stress testing, that attempt to check how the system will respond in the real world.

Stress testing is particularly valuable in checking what happens when peak loads are reached or exceeded, when ranges are exceeded or timing is incorrect, or operators

continued on page 2

"Software Testing..." continued from page 1

and maintainers take "shortcuts" or make mistakes. It was a skilled operator shortcut that triggered the Therac-25 overdoses. In one example of operational testing cited in ***Operational Test & Evaluation***, ©1986, by Roger T. Stevens, two skilled systems personnel pushed unanticipated combinations of buttons and uncovered over 300 software errors. Actually, most such errors are uncovered by using ***unskilled*** operators.

One test area that is often overlooked is the software or system response when hardware fails. This is especially unfortunate since most modern systems tend to rely on software to maintain safety—"fail-safe" or "fail-soft"—when a hardware component fails. Such software tends to be poorly tested, if at all, because to do so may require destructive hardware testing and/or use of very expensive simulation.

Finally, because a major system failure due to a particular combination of lesser failures, faults, errors, and/or events, including personnel actions, is rare and testing resources are limited, multiple failure testing almost never occurs. Moreover, if we can assume that any particular combination is worthy of being singled out for testing, we can alter the design to specifically guard against it. That may be cheaper!

Yet nearly every catastrophe of recent years was due to multiple failures and extraordinarily unlikely events—often involving failures of several safety mechanisms or backups. [N.B. While the probability of any specific rare sequence may be vanishingly small, the probability that ***some*** such sequence will occur is very high!] Before occurrence, the estimated likelihood of the Three Mile Island event was less than once in 1,000,000 years. The Chernobyl event was estimated to be far less probable than that. In the latter case, what the designers didn't foresee was the technical ignorance of key operational personnel, that all of the automatic safety mechanisms would be disabled because a test of emergency procedures had been authorized(!), and that the last reactor manual safing mechanism had a design flaw which provided the final element needed to trigger the accident.

Since complex software systems can only be partially tested at best and this is aggravated by schedule pressures and limited resources, lessons learned on previous systems should be used for developing a practical testing strategy. The best approach is to focus on potentially high risk areas and code critical modules first so they are available for testing longer. Another good guideline is to base testing decisions on what is truly ***important***. What is high risk or important can be very far from obvious. For safety-critical systems, the identification of such functions should be based on hazard and technical risk analyses, statistical sampling techniques, and on user surveys, which will be discussed in other articles.

Implicit in all of this is the notion that test sequences must be carefully planned. Experience on large projects has shown that effective ***test planning*** is the most significant phase of the testing effort. Recording test planning decisions is also important. Documentation should include not only what will be tested, but also what will ***not*** be (or has not been) tested. Typically, the latter is not done. Such documentation is particularly valuable for systems that are expected to be periodically modified and retested after delivery.

Testing should be planned as a continuous process over the entire product life cycle. When a software fault does slip through a well-disciplined development and testing process, it is cause not only to fix the software fault, but also to investigate and fix the ***test planning*** or ***testing*** process fault which allowed the software fault to slip through. This ensures that quality is built into the software up front and can be maintained over subsequent product improvement cycles.

These guidelines apply to any computer-based system, but the testing approach for safety critical systems must go beyond this. More on this in the next issue. ■

OT&E PROCESS IMPROVEMENT

by Natalie Reed, ACT-200

As part of the William J. Hughes Technical Center CMM-based Process Improvement Initiative, the ATC Engineering and Test Division, ACT-200, is working with Crown Communications, Inc., and Software Engineering Technology, Inc., to develop a process improvement plan for OT&E processes. The plan will be based on an internal needs analysis, a CMM assessment, and a review of best practices in OT&E.

As FAA moves further into the flexibility afforded by the new Acquisition Management Strategy, it will be essential that a standard, tailorable process exist for OT&E (as well as other key FAA processes). **A process must be defined and under control before it can be systematically improved**. A baseline in process performance permits ongoing comparison between current and historical performance, and between business-as-usual and pilot approaches.

In this ACT-200 project, the standard activities and work products associated with OT&E are being defined in terms of Objectives, Participants, Entry Criteria, Tasks, Measures, Verification, and Exit Criteria. Once the current process has been accurately documented, specific improvements will be planned. Improvements will be considered based on current concerns, a recent independent CMM-based assessment, and best practices in industry and other federal agencies. ■

*inter*FACE

is published quarterly by SEPG

DOT/FAA/AIT-5
800 Independence Avenue, SW
Washington, DC 20591

Chief Scientist for Software Engineering
Floyd Hollister (202) 267-8020

Editor
Norm Simenson (202) 267-7431

FAX (202) 267-5080

SOFTWARE TEST ENGINEERING

Carmen Trammell and Jesse Poore
Software Engineering Technology, Inc.

There are many ways to drive a person crazy. One way is to expect him or her to handle the combinatorial explosion (i.e., to develop software test plans) on the basis of personal experience and common sense alone.

- For software systems in which the number of inputs in a usage session is unbounded, it is impossible to test all scenarios of use because the number of scenarios is infinite. (Assume a system with 2 possible stimuli, A and B. Possible scenarios of use: A, B, AA, AB, BA, BB, AAA, AAB, ABA, ABB, BAA, BAB, BBA, BBB, AAAA...ad infinitum)
- For software systems in which the number of inputs in a usage session is bounded, it is impractical to test all scenarios of use because the number of scenarios is astronomical. (Assume a system with 20 possible stimuli allowing a maximum of 10 inputs in a usage session. Possible scenarios of use: sessions of length 1 + sessions of length 2 ...+ sessions of length 10 = 10,778,947,368,420.)

There is help.

Engineering is the application of science to produce useful artifacts. Software engineering is the application of science to produce useful software. So, what is the relevant science for software testing?

- When a population is too large to study, all one can do is study a sample.
- All software testing is based on sampling.
- Statistically correct sampling allows valid inferences from testing to operational use.

The appropriate science for software test engineering is applied statistics. **DON'T STOP READING**. Applied statistics is not what you think. It is not the arcane field that it may have seemed to you in college. It is a world of useful formalisms that make it possible to answer questions about populations that are too large to study exhaustively.

Test Design- How can I choose the critical variables to apply in testing from the many variables that influence operational performance?

Rx- Use a combinatorial design—a "fractional factorial" experimental design—to vary several factors at once. Or, use multi-dimensional scaling and factor analysis to determine which factors vary together and can be combined in test design.

Optimal use of resources- How can I plan testing such that the maximum value is gained from limited test resources?

Rx- Use Operations Research (OR) techniques to define constraints and objectives, and to optimize test plans for the objectives subject to the constraints. While the application of OR to software engineering is still new, there is literature available for the beginner in OR, and there are some software tools which can help.

Safety- How will this software perform under hazardous usage conditions?

Rx- Represent hazardous usage in a Markov model of operational use, and apply standard Markov calculations to determine the probability of hazardous event X in a single usage session, the long-term occurrence rate of hazardous event X, the average number of uses between hazardous event X, the probability of a given sequence of events, and other performance characteristics of the software under hazardous operating conditions. [***editor's note***: This can also help design. If hazardous event X turns out to be too probable for comfort, it is generally easy to insert a "guard" into the design to block any path to it. This prevents any known way to generate event X. To guard against the impact of event X, should it occur by way of an unknown path, additional safeguards can be inserted to detect its occurrence and to limit its impact.] The appropriate Markov calculations are made relatively easily by using any of a number of matrix algebra CASE tools. [***editor's note***: Monte Carlo simulation tools provide a powerful way to simulate any Markov process and generate statistics painlessly with as little statistical error as you wish, depending on the number of trials used.]

Reliability- What is the expected reliability of this software under given usage conditions (routine, non-routine, safe, unsafe, mature, immature, etc.)?

Rx- Use a stratified sampling strategy for random statistical test case generation, and determine the levels of reliability and confidence that may be

continued on page 7

Letter from the EDITOR

ERROR-FREE SOFTWARE

Why can't we improve our software development processes to the point where—using formal methods, for example—we can develop (nearly) error-free software? Well, we may actually approach this nirvana for deterministic errors, but I would guess that the probability that we will do as well for non-deterministic errors is zero. (See "Heisenbugs in the System" by Myron Hecht et al for an explanation of deterministic and non-deterministic errors.) We are increasingly living in a world where multi-tasking, multi-processing, multi-threaded, fully distributed software is running on multiple processors using fully democratic (or nearly so) interprocess protocols.

The possibilities of "glitches" in such systems are already astronomical and are increasing rapidly! Our best defense is to build-in fault tolerance which increasingly relies on "expert systems" intelligence or artificial intelligence to figure out what has gone wrong, when it inevitably does, and try to arrive at a "rational" strategy to fail-soft or fail-safe. Obviously, because of cost, this is possible only for the most critical software. For the rest, we must rely almost exclusively on testing—including the much maligned "beta testing."

This puts a premium on developing architectures which can identify a critical kernel requiring the most stringent and costly techniques for development and assurance against failure. All the rest can make do with less costly techniques. I can get WINDOWS 95 for less than $200. But it still crashes under heavy use about once a day. It has hundreds of known anomalies, some which Microsoft has no intention of fixing. Would it be worth it to reduce the bug rate by 60% if it raised the price to $200,000 per copy? We need to be able to put a dollar value on system and subsystem availability and produce accordingly.

Norm

HEISENBUGS IN THE SYSTEM

by Myron Hecht, Herbert Hecht, Jady Handal, and Dong Tang

COTS systems have the potential of significantly reducing lifecycle acquisition and operating cost, and may also reduce program developmental risk. As a result, they have become an increasingly important part of large real-time systems. However, as discussed, it is also necessary to ensure that only suitable COTS components are integrated into mission critical systems, and suitable COTS components may not be available at an acceptable system lifecycle cost.

The drawbacks of COTS components and systems for safety critical applications are unknown reliability and unknown failure characteristics. A related problem is that a high availability requirement normally dictates a fault tolerant system design to provide for the needed redundancy and reliability. Few COTS come so equipped.

Assessing the reliability of COTS software is generally the most important issue associated with the use of COTS components or systems. Traditional analytical reliability methods were developed for **random hardware** failures and are no longer adequate. A satisfactory model of software failures is essential to a system-level characterization of both reliability and availability. Unfortunately, no generally accepted methodology of software reliability ***prediction*** has emerged.

But, while software reliability ***prediction*** methodology remains questionable, software reliability ***measurement*** methodology **is** currently satisfactory. To a large extent, software reliability measurement can use the same concepts and theory as system reliability life testing. Such reliability life testing and measurement have a long and established history in the nuclear, aerospace, and DOD communities.

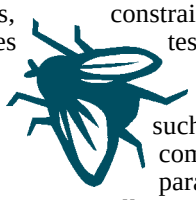
The underlying assumption in these measurement-based approaches is that the fundamental failure mechanisms are triggered non-deterministically. Such failures result in an unpredictable crash, hang, or processing slow-down which prevents the system from running successfully to completion. Causes can most often be traced to transient, unpredictable, and infrequently occurring conditions related to timing, loading, or sequencing of events and/or processes. (Because of the uncertainty in their occurrence and the difficulty of reproducing such failures, they are frequently called "Heisenbugs," an evil pun based on the name of the physicist who won fame by formulating the uncertainty principle in quantum mechanics.)

However, there is another class of failure in which the software produces a consistently unacceptable output as a function of the same inputs and process states, e.g., an air traffic conflict probe algorithm may fail to detect a future separation violation. This **deterministic** failure may often be traced to a logic fault in the code or algorithm, or an incorrect input value provided to the code. The root cause may be due to defects in the system or software requirements.

The techniques and methodologies for estimating the probabilities of deterministic system/software failures are immature. It is tempting to assume that an adequate testing program should uncover them. But testing resources are finite, and it is generally impossible to provide sufficient time or money to perform the level of testing needed to uncover all such failures, even in systems designed for high dependability. There are serious deficiencies even in systems and software which use a formal requirements specification language.

Therefore, when estimating software failure rates, one must look at both non-deterministic **and** deterministic failures. If the proportion of deterministic failures at the final stages of testing or integration, or in initial operation, is high, then reliability predictions made exclusively on the basis of non-deterministic failures is likely to be poor.

Fault tolerance is generally implemented using redundant subsystems, and software which provides services for duplicated transmission on networks, health checking of the active and redundant subsystems, synchronization of data, and similar functions. These services are far removed from the normal application function. For COTS components or systems, they may have to be provided either by the application system developer or by a third party vendor.



Assuring that the fault tolerance provisions really work is extraordinarily difficult. The challenge at the highest level is to show that availability requirements are being met. More detailed quantitative testing can identify errors masked by the proper action of the fault tolerant function, system bottlenecks, or potential hazards. This can provide insight for requirements or design corrections.

A key parameter in assessment of a fault tolerance system is the probability of successful recovery given that a failure has occurred. This parameter, often called ***coverage***, can best be determined through a regime of highly focused testing driven by thorough system analysis, supplemented by continuous data collection designed to capture spontaneously occurring failures. Such testing is different from the "failure recovery" testing which is usually performed as part of system acceptance testing.

In failure recovery testing, failures are usually simulated by disconnecting communication lines, failing individual software tasks, or powering components down. However, in distributed processing software systems, it is far more likely that transient (interprocess) communications errors may occur, or that one or more tasks running in one or more processors may crash, hang, or otherwise suffer degraded performance. Simulating such failures generally requires specially developed software within the running system supplemented by additional hardware and software to capture and analyze this data. The detailed techniques, which often go under the name of "fault injection," are described in the literature (e.g., by the authors). Quantitative analyses of fault injection testing have been developed.

Obtaining adequate data from which to assess reliability and availability is critical, but can be very difficult to implement in practice. It can significantly add to the design complexity (and cost) of operational software, or to the constraints of an already expensive testing program. Adequate data collection means monitoring and recording all events of interest such as failures and recoveries of components, and performance parameters for the target system in all operational regimes. Data must be collected on all failure modes so that an assessment of the importance of deterministic failures can be made.

Measurements must be made continuously for a sufficient period to yield statistically significant data. Operating logs should include information about the location, time, and type of the error, the system state at the time of fault detection or operational failure,

continued on page 7

EARLY TESTING

by Tom Gleason

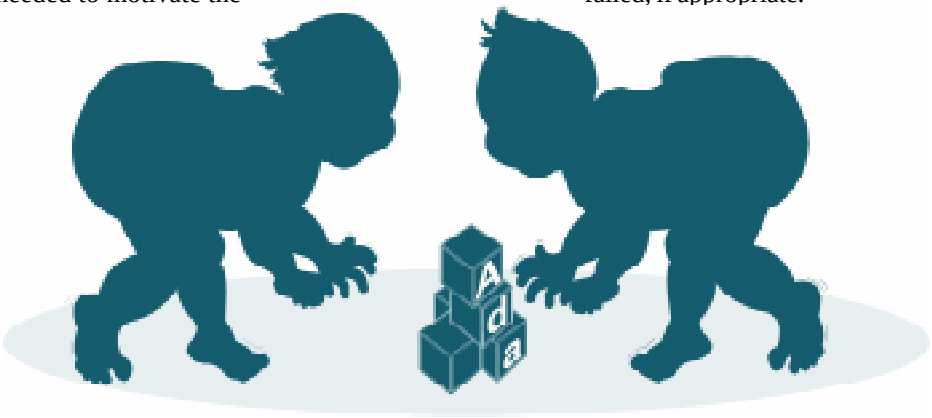
As a Quality Assurance Specialist for the government at Loral Air Traffic Control in Rockville, I believe I have gained some insights into the implementation process that are worth sharing and will contribute to reduced risk during the implementation phase.

While there are many activities which can be used to validate requirements and to verify a design developed from a set of requirements, unit level testing and peer review are the first line of defense for the implementers. Although they are primarily intended to identify problems in design implementation, they can also do an excellent job of catching errors in design and requirements. String or thread testing of several units (or a full object) to verify one or a few functions or capabilities can be seen as an intermediate step between unit and full integration test which is especially apt to identify design and requirements errors. The risk at this level is that inadequate testing or review will allow many errors to slip through. Ideally, all requirements, design, and implementation errors should be caught before full integration testing; otherwise, schedules and cost are likely to be seriously impacted. The objective of integration testing is ***not*** to debug the system, but to verify that there are no unexpected interaction problems when all of the functional threads or objects are brought together and made to play in the same arena.

Unit testing is planned to verify that a unit (or component) performs its intended design—that all outputs and state changes are achieved as specified over the range of inputs via interfaces and internal state data. Verifying the correct functioning of a unit is ***not*** intended to be performed through an exhaustive set of tests which exercises every possible combination of inputs and outputs. Unit testing ***is*** intended to provide an objective demonstration of the correctness of the unit and to exercise all code and key states of the unit (but not necessarily every path). Unit test cases are developed and evaluated using these criteria.

Risk may be mitigated with respect to implementation by the development of unit test plans and unit test cases which adequately reflect design specifications and user requirements, the key to which is peer review. Unit test plans are needed to motivate the programmer to think ahead about testing. Early on, the programmer must identify external dependencies and the level of effort needed to perform unit tests. A good programmer will implement a unit so that it is easy to access key state data during test or for other purposes. Unit test cases should make use of the best knowledge of software engineering and test issues in general, such as testing at boundary conditions, and specific knowledge of the unit. Peers can greatly assist by bringing the knowledge of the programmer's community to bear on both the general and specific issues. Peer review should be approached as a game. The winner is the one who, by general agreement, contributes the most in producing a unit test which achieves the maximum testing thoroughness with the least effort. People should be rewarded for figuring out clever ways for "breaking the code" or for spotting bugs in the code.

Once a unit is through testing, both it and its unit test should be placed under configuration control. As later errors are discovered and corrected in a unit, and change instruments (SPRs/PTRs) are opened and closed, changes to test cases must not be overlooked. A change in the code may require some change in the unit test to reflect a changed result or to add a previously omitted test case. The unit test must be considered a part of the unit and must always be an adequate test of the current as specified unit. Risk is mitigated by assuring that the test cases, as well as the code and comments, always reflect the current design specifications. Change documentation must include some indication that the unit test has been separately reviewed, and corrected if necessary. For future defect prevention analysis, it may include something about why the test failed, if appropriate.



Peer reviews are the best way to assure the adequacy and correctness of test plans and cases. Peer reviews make sure design specifications are properly developed from the user requirements and are properly implemented. After the unit designer and programmer, peers know the most about the specific problems of the design and implementation and bring the most knowledge to the unit development process.

The FAA quality reliability officers (QROs) need to be actively involved in a contractor's peer review process for test, which is fundamental to the quality of the product produced and critical for assuring product safety. This should include attendance and participation in requirements reviews and analyses, design reviews and analyses, unit test plan reviews, test case reviews, and code inspections. It means that the QRO must maintain a high level of technical competence to be effective in assuring that peer reviews accomplish their risk reduction goals. Adequate training is absolutely essential to maintaining skills in this area. The QRO participation needs to be viewed as a plus by all levels of the contractor. ■

LAST LAW OF PROGRAM MANAGEMENT

We have plans for every conceivable contingency. What could possibly go wrong?

Regulating the Critical System

by J. Scott, G. Preckshot, G. Johnson, Lawrence Livermore National Lab

A critical system is one whose failure or imperfect operation may produce significant harm, including death or destruction, to people or property. There are few, if any, special procedures or processes which can be used to assure the correct operation of such systems, but everyone involved at every stage of their life cycle must take every step possible to prevent failure. While a little trite, it nevertheless bears repeating: the price of safety is constant vigilance. Regulators have a special role in assuring constant vigilance of such systems.

The system life cycle for digital systems includes activities involving hardware, human factors, and software arranged into three, more or less parallel, but often converging life cycles. They share a common system architecture and design, and converge for system test, installation, acceptance, and post-acceptance activities.

It is currently impossible to demonstrate conclusively that a digital product meets all of its quality and performance requirements. Many assurance techniques must be used throughout the system life cycle to achieve high confidence in the safety of a product. As a practical matter, no single activity, such as system or software testing, can provide this level of confidence. It is important that the assurance techniques be effective: some time honored techniques, such as a long history of fault free operation of a system in a relatively restricted environment tend to confer a high level of confidence but are, in fact, almost valueless. 10,000 repetitions of the same test are generally far less useful than one repetition each of 10,000 different tests.

Software testing activities are a subset of software verification and validation (V&V) activities which, in turn, are a subset of overall system dependability measures. Other software V&V activities complement and influence software testing. See IEEE 1012-1986 for an overview of software V&V activities.

Dependability is a system issue. Planning for dependability begins with the selection of activities calculated to detect and avoid faults in every phase of the life of the system. The architecture chosen and the design implemented must provide for the mitigation of anticipated failures through techniques such as redundancy, diversity, and fault-tolerant designs in which software plays a major role. While particular errors and failures cannot be predicted (if they can, it is easy to provide design compensation), classes and categories can be. These, or more properly their effects, are guarded against by both built-in safeguards and external procedures, such as testing.

For example, if system elements are reused, a component acceptance process commensurate with target component reliability and target system dependability goals must be developed. This should include the plans and commitments for achieving dependability. Like other product assurance processes, the component acceptance process cannot rely on a single technique.

All participants in the development process should understand system architectural and design choices, and their implications. The regulator should assess the impact of these early, crucial choices on the target system's dependability. During the initial stages of architectural selection and top level design, the regulator should also plan for subsequent reviews of selected process and activity records to ensure that planned activities are properly implemented. The regulator should also ensure that the architecture/design will provide other evaluators of the system, such as testers, with adequate access to system operational parameters which can provide assurance that all parts of the system are operating as designed under every conceivable operating/testing condition.

Software testing is related to every aspect of the system life cycle. The allocation of system requirements to software, the system hazard analyses, and ensuring that the design provides adequate access to operational data are all early activities important to the testing program. Requirements allocated to software are the basis for tests of essential functions and performance. System hazard analyses generate information on external threats and abnormal conditions under which system dependability must be verified. This information is used to develop test cases which address performance in off-normal situations.

Proper system design choices can help to limit the types of testing needed or restrict the need for intensive software testing to a few modules. On the other hand, some choices can constrain the methods and approaches available for use in downstream testing—sometimes to the point of making the system untestable or making the tests prohibitively expensive. System integration, installation, acceptance, and operational tests involve all aspects of the system: software, hardware, and human factors. Careful planning and lots of communication among all participants will help to make these tests maximally effective.

The role of software testing in the overall system dependability effort is to verify software functions and performance and to assure robustness in the face of abnormal conditions and events. Exhaustive software testing is usually not practical or practicable—testing cannot prove the absence of faults. While we can hope that software improves as faults are detected and removed, the reality is that the fault removal process is likely to introduce new faults. So, testing can never be a substitute for choosing a fault tolerant architecture, designing quality into a product, or for stressing quality software production techniques. Even formal methods have severe limitations. Analytical models generally do not exist to predict the performance of software, which exhibits strong properties of non-linearity. Indeed, software behavior often shows patterns of chaos theory: minor differences in inputs or initial conditions can produce disproportionately large differences in outputs or end conditions. Therefore, it is very difficult to achieve high confidence in the software through testing alone.

Many methods and strategies exist to maximize the effectiveness of software testing, e.g., testing at different levels (unit, integration, and system), using a combination of static analysis, structural testing, functional testing, stress testing, and statistical testing. Since each strategy works best with certain types of faults, a comprehensive testing program combines different strategies at different testing levels.

For product assurance, a regulator is typically limited to auditing quality assurance records of the results of testing, but the regulator must also assess the adequacy of the choice of test strategies and range (coverage).

continued on next page

"Regulating the Critical System"
continued from previous page

This assessment focuses principally on the critical aspects of the application and the degree to which the proposed testing strategy and coverage will verify those qualities upon which regulatory approval depends. The regulator must anticipate which tests are most important in satisfying safety concerns. This means that a knowledge of the system architecture and the results of hazards analyses are crucial for deciding which tests help demonstrate critical safety attributes.

Since regulators work indirectly with artifacts of a test process, solid documentation of all aspects and elements of the test process and a solid configuration management process is crucial for regulators to carry out their function.

A compendium of LLNL work on software processes for high integrity systems is available at:
http://nssc.llnl.gov/FESSP/CSRC/CSR.html

The report,
A Proposed Acceptance Process for Commercial Off-the-Shelf (COTS) Software in Reactor Applications, *an appendix,*
Testing Existing Software for Safety-Related Applications,
should be of particular interest to readers.
Please contact Gary Lynn Johnson, ***johnson27@llnl.gov****, or the authors, for more information.* ■

"Software Test ENGINEERING"
continued from page 3

inferred for the software in field use from the size of the test sample. Again, there is help in the technical literature.

Retesting- How can I determine how much testing needs to be done after a change in code?

Rx- Use the statistical approach to "combining information" across the life cycle to determine how much new data is needed to bring the new version of the software up to the levels of reliability and confidence of the previous version.

Everybody can't know everything. Aviation specialists, software engineering experts, and statisticians need to work together to address the monumental challenges in software testing. Too many test professionals have only their experience and common sense to use in the awesome task of software testing. Experience and common sense are surely necessary, but they are far from sufficient. The FAA mission to ensure public safety will be best served by an approach to software testing that is based on sound science. ■

LAW OF SCALE

If "scaling up" were indeed as simple as many people make it out to be, then the Master Engineer would have designed us all as giant amoebas.

"Heisenbugs in the System" *continued from page 4*

some history (i.e., sequence of immediately preceding states), and error recovery information where applicable. This normally requires built-in instrumentation of operational software components, as well as special instrumentation packages associated with exceptional or recovery event sequences.

Measurement based assessments are not a replacement for other aspects of the system engineering process. For example, system level requirements should specify minimum COTS performance, such as response times or maximum loads, and functional capabilities. Maintainability requirements should identify essential monitoring and maintenance management capabilities. Quality assurance requirements should address the vendors' system and software development and/or integration process maturity, and legal protections, such as source code escrow. Safety requirements may impose severe constraints on COTS failure modes and failure detection properties, or the need for expensive backup equipment. Finally, test and evaluation must be capable of assuring that all requirements allocated to the COTS are met. These additional constraints and controls are necessary to ensure that selected COTS and NDI components may be safely incorporated into ATC systems.

J. Handal (AND-8)
may be contacted at the FAA,
202-267-3241

M. Hecht, H. Hecht, and D. Tang
may be contacted at
SoHaR Inc. 8421 Wilshire Blvd., Suite 201, Beverly Hills, CA 90211-3204 ■

Training Opportunities...

The FAA SEPG has developed a training program consisting of the following topics. Classes are to be offered periodically throughout the year. Please contact your organization's SEPG member for schedule and enrollment information or discussion of your software training needs.

- **Capability Maturity Model (CMM) for Software and Associated Key Process Areas for Level 2**
- **CMM for Software Acquisition and Associated Key Process Areas for Level 2**
- **People CMM**
- **Open Systems, the Promises and the Pitfalls**
- **Software Capability Evaluation Training**
- **Software Risk Management**
- **Requirements Management**
- **Metrics**
- **Clean Room**
- **Software Development Cost and Schedule Estimation**
- **MIL-STD-498, Use and Tailoring Opportunities**

In This Issue

interFACE

Newsletter of the Software Engineering Process Group

Volume 5, Number 4
November 1996

SEPG
FEDERAL AVIATION ADMINISTRATION

Conference Calendar

Software Engineering Laboratory (SEL) Software Engineering Workshop
December 4-5
Code 552
Goddard Space Flight Center, Greenbelt, MD 20771
Contact: (301) 286-6347

Software Technology Conference
April 27-May 2
Contact: (801) 521-9055 or (801) 521-2822

Society for Software Quality (SSQ) Meeting

December 10	Rick Hefner, SE Technology: Year in Review, Maryland
January 14	Mark Servello, Why to NOT have a Software Assessment, Virginia
January 27	Day Long Roundtable Network Security, Maryland

Contact: Chris Dryer (202) 767-2894

Federal Software Process Improvement Working Group (FEDSPIWG)
Held Monthly at NOAA

DOT/FAA/AIT-5
800 Independence Avenue, SW
Washington, DC 20591